Transparent Query Caching in Peer-to-Peer Overlay Networks

Sunil Patro and Y. Charlie Hu
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
{patro, ychu}@purdue.edu

Abstract

Peer-to-peer (p2p) systems such as Gnutella and KaZaa are routinely used by millions of people for sharing music and many other files over the Internet, and they account for a significant portion of the Internet traffic. The p2p traffic can be broken down into two categories: protocol messages for maintaining and searching the overlay p2p network, and data messages for downloading data files. This paper makes the following two contributions. First, we present a study of the locality in the collective Gnutella query protocol messages being initiated and forwarded by servents within the same organization. Second, we propose a transparent query caching scheme to reduce the bandwidth consumed by p2p traffic going in and out of the gateway of an organization.

Our locality measurements show that there is significant locality in the collective queries going through a gateway forwarded by servents behind that gateway, and the locality increases with the population of those servents. Our proposed transparent caching scheme preserves the user experience, i.e., users continue to use the same servents as before, and queries will result in similar query hits with or without the caching running. Measurements of our transparent caching proxy in an experimental testbed of eight passive Gnutella servents in a LAN has shown a query cache hit rate of up to 38%, an uplink query traffic reduction of up to 40%, and a downlink query hit traffic reduction of up to 12% at the gateway.

1 Introduction

In the first six years of the Internet explosion, one type of dominating traffic over the Internet had been HTTP traffic from accessing the World Wide Web. Around the year 2000, a new paradigm for Internet applications emerged and has quickly prevailed. Today, the so-called peer-to-peer (p2p) systems and applications such as Gnutella and KaZaa are routinely used by millions of people for sharing music and other files over the Internet, and they account for a significant portion of the Internet traffic. For example, the continuous network traffic measurement at the University of Wisconsin (`http://wwwstats.net.wisc.edu`) shows that peer-to-peer traffic (KaZaa, Gnutella, and eDonkey) accounts for 25–30% of the total campus traffic (in and out) in August 2002, while at the same time, web-related traffic accounts for about 23% of the total incoming and 10% of the total outgoing traffic.

To reduce the bandwidth consumption and the latency in accessing web content, web caching has been extensively studied. Today, web caching products are offered by numerous vendors and widely deployed in the Internet, and have shown to be highly effective in reducing network bandwidth consumption and access latency. In contrast, little progress has been made towards caching p2p traffic. The traffic generated by a p2p network such as Gnutella falls under two categories: the protocol messages for maintaining the overlay network and for searching data files in the overlay network, and the actual data messages for downloading files. Previously, researchers have found that search messages (queries) in Gnutella networks exhibit temporal locality, suggesting that caching should prove to be an effective technique in reducing network bandwidth consumed by these queries and the latency in retrieving query replies. Several query caching schemes have been developed by modifying Gnutella servents, and have confirmed the effectiveness of query caching. However, these previous schemes have several limitations. First and most importantly, these schemes were proposed in the context of modifying Gnutella servents, and rely on the wide adoption of such modified servents to achieve the benefit of caching. The wide adoption of any modified servents is a potentially slow process. One simple reason for this is that users are used to the particular servents they have been using, because of their specific functionalities, GUIs, dependence on particular operating systems, and so on. Second, all the previous proposed caching schemes were implemented at each individual ser-

vents, and thus they cannot exploit the locality in the queries collectively forwarded or initiated by a set of servents, for example, within the same organization.

In this paper, we explore query caching schemes in p2p overlay networks that overcome the above mentioned limitations. Our overall approach is to develop transparent p2p caching proxies that will be attached to the gateway routers of organizations or ISPs, i.e., similar to how web caching proxies are typically deployed, with the goal of reducing p2p traffic in and out of the gateways. The gateway Cisco router (over 80% of routers in the Internet are Cisco routers) running the WCCPv2 protocol will be configured to redirect TCP traffic going to well known p2p ports, e.g., 6346 for Gnutella, to the attached p2p caching proxies.

We focus on Gnutella networks in this paper. The specific contributions of this paper are as follows:

- We study the locality in queries forwarded by servents inside a gateway and its implications to the effectiveness of caching at the gateway.
- We present a transparent caching scheme that preserves the p2p user's experience.
- Since Gnutella servents often use the same TCP port for both protocol and data traffic, we also present a scheme for integrating our query caching proxy with any off-the-shelf high performance HTTP caching proxy without reimplementing one into the other, and without incurring significant overhead.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the Gnutella protocol. Section 3 presents the measured locality in the collective Gnutella query messages observed at the gateway of an experimental testbed consisting of eight Gnutella servents in a LAN. Section 4 presents and evaluates our query caching scheme. Section 5 presents a scheme for integrating our transparent query caching scheme with HTTP data caching for Gnutella networks. Finally, Section 6 discusses related work, and Section 7 concludes the paper.

2 The Gnutella Protocol

The Gnutella protocol [1, 2] specifies the operations for maintaining and searching a peer-to-peer overlay network superimposed on top of the Internet. Gnutella peering nodes (called *servents*) connect with their direct neighbors using point-to-point connections. In the following, we first describe how a new node joins an existing Gnutella network, and its implications on the topology formed by the servents within an organization. We then briefly describe the protocol for the query request and response messages, and data file downloads.

The joining process and its implications on topology We describe the joining process of a typical Gnutella servent. Any host running a servent can connect to the Gnutella network. When a servent wants to connect, it first looks up its *host cache* to find addresses of Gnutella servents to connect to. The servent addresses are removed from the host cache after they are read. If the host cache is empty, then it tries to connect to a well known Gnutella *host cache server* (also called PONG server) to receive PONG messages in response to its PING message. After receiving the PONG messages, the servent tries to establish connections with the servents whose addresses are obtained from the PONG messages.

In a typical Gnutella servent, after the pre-specified number of Gnutella connections is established, the servent sends PING messages periodically to monitor those connections and in response receives a number of PONG messages[1], which are appended at the end of the host cache. In addition, once an existing connection with some servent is broken down, the servent's address information is saved and eventually will be added to the host cache when the servent leaves the Gnutella network.

In summary, during the joining process of a typical Gnutella servent, the neighbors are chosen from the host cache whose content is fairly random. This suggests that it is unlikely servents from the same organization will become neighbors of each other, let alone forming a clique, and consequently, the query messages will unavoidably travel across the gateway of the organization.

Query requests and responses In order to locate a file, a servent sends a query request to all its direct neighbors, which in turn forward the query to their neighbors, and the process repeats. When a servent receives a query request, it searches its local files for matches to the query and returns a query response containing all the matches it finds. Query responses follow the reverse path of query requests to reach the servents that initiated the queries. The servents along the path do not cache the query responses.

To avoid query requests and responses from flooding the network, each query contains a TTL (time to live) field, which is usually initialized to 7 by default. When a servent receives a query with a positive TTL, it decrements the TTL before forwarding the query to its neighbors. Queries received with TTL being 1 are not forwarded. In other words, queries are propagated with a controlled flooding.

It is easy to see that the same query can visit a servent more than once during the controlled flooding, i.e., through different neighbors of that servent. To make sure that each node does not serve the same query more than once, each query is identified by an unique identification called `muid`.

[1] PONG messages are generated by a servent either on its own behalf or on behalf of other servents using its PONG cache as supported in Gnutella version 0.6.

When a servent receives a query with a `muid` it has encountered in the past, it will simply drop the query. Obviously, such repeated copies of the same query should not be counted when measuring the locality among a set of queries.

Data file downloads After the query hits are received by the servent, the user can select a file to download. This download request is carried out through an HTTP GET message sent over a direct TCP connection between the requesting servent and replying servent.

3 Locality Measurement

In this section, we study the locality in queries forwarded by a set of servents behind a gateway and its implications to the effectiveness of caching at the gateway.

3.1 Methodology

To monitor the collective traffic going out of a set of servents behind a router, we implemented a tunneling probe running on a FreeBSD machine that is configured as shown in Figure 1. First, the probe machine is placed next to the gateway Cisco router running WCCPv2. The gateway router will be configured to redirect all packets destined to known Gnutella ports (for example, 6346) to the attached tunneling probe. Second, the probe machine is configured to use IP forwarding to redirect all TCP traffic going to other destinations with port 6346 to port 6346 of localhost, on which the probe is listening. This allows the probe to hijack all outgoing connections with destination port 6346. The probe then initiates a TCP connection to the original destination, and tunnels all the packets traveling towards both directions, Thus to outside destination servents, the probe machine appears as the servent that initiated the connection, while the servents inside the gateway that initiated the TCP connection think they have established a TCP connection with the destination servent. In other words, the hijacking of the TCP connection is transparent to the servents inside the gateway, but not to servents outside the gateway.

While tunneling the Gnutella traffic, the probe collects the information about the packets being tunneled through, such as the type of the messages, the size, and in terms of query, the TTL, and the frequency. These information will then be used to analyze the locality in the query requests forwarded or initiated by all the servents inside the gateway as well as data file downloads if they use the same connections, i.e., with port 6346, to the destination servents.

To monitor the Gnutella traffic going through each individual Gnutella servent, we made modifications to `gtk-gnutella`, a UNIX-based open-source servent for Gnutella, to record all the protocol messages and data downloads in and out of the servent.

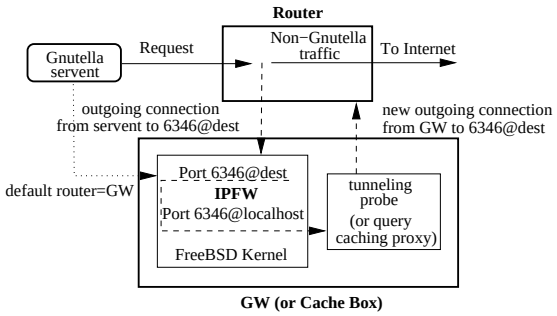

Figure 1. The tunneling probe running on a PC with IP forwarding (called a GW) hijacks servents' outgoing connections. The router can be absent if the servents are configured to use the GW as their default router. The caching box discussed in Section 4 replaces the tunneling probe with the query caching proxy.

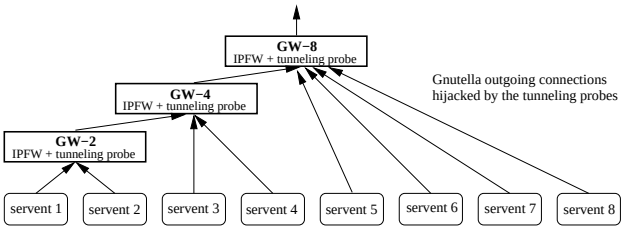

Figure 2. Configuration of the measurement testbed. GW-2, GW-4, and GW-8 hijack outgoing connections (with destination port 6346) of Gnutella servents, and appear as the origin of connections to outside servents.

To simplify the measurement setup, instead of using a real router, we configured each servent to use the GW as the default router (see Figure 1). IP forwarding rules are specified on the GW such that packets going to port 6346 of any destination will be forwarded to port 6346 of localhost, and all other traffic are forwarded. Thus only outgoing Gnutella connections will be hijacked by the tunneling probe.

3.2 Measurement Results

We measure the locality among a set of eight passive servents running on a cluster of eight PCs behind a router. Each servent is configured to allow four incoming and four outgoing connections. Each servent is passive, because it only forwards queries and query hits, but does not initiate any queries. Furthermore, it does not store any files for sharing. To measure the locality among queries from varying numbers of servents, we configured three PCs as GWs running the tunneling probe, and we configured the whole testbed into a hierarchy as shown in Figure 2. GW-2, GW-4, and GW-8 are the three GWs, and traffic on the outgoing connections of two, four, and all eight servents go through GW-2, GW-4, and GW-8, respectively.

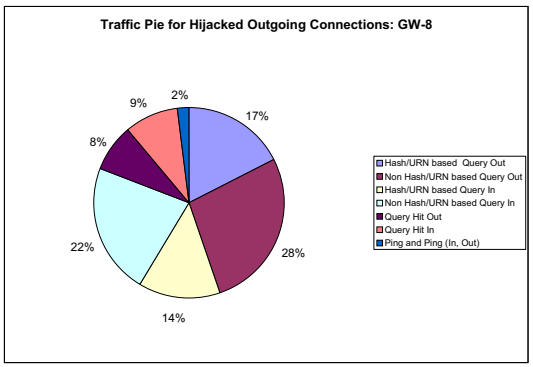

Figure 3. Percentage distribution of different types of incoming and outgoing messages at GW-8 on hijacked outgoing connections.

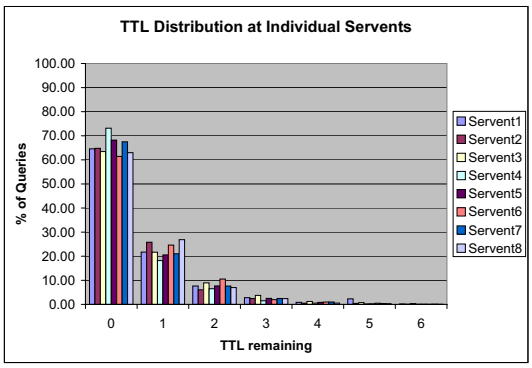

Figure 4. TTL distribution of queries going into the servents on their incoming and outgoing connections, excluding empty queries.

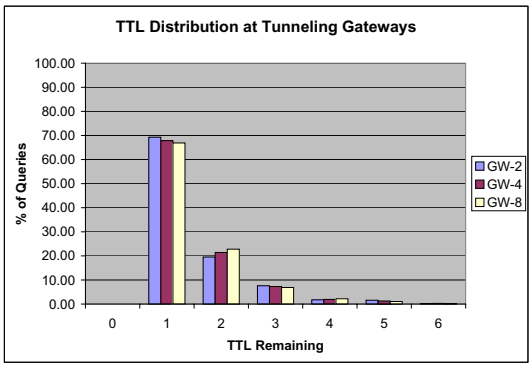

Figure 5. TTL distribution of queries going into the gateways on hijacked outgoing connections, excluding duplicated queries with the same `muid`. and empty queries.

We started the experiments with all eight servents at 1:00am EST on January 13, 2003 (after a 15-minute warm-up period), and the experiment lasted for an hour. Each servent recorded all messages sent and received, and the tunneling probe recorded all Gnutella packets going into and out of the GWs on hijacked outgoing connections.

3.2.1 Traffic Breakdowns

We first report the total Gnutella traffic going through each servent and GW and their breakdowns into different types of messages. Figure 3 shows the percentage distribution of different types of both outgoing messages and incoming messages on outgoing connections hijacked at GW-8. The same breakdowns at individual servents, at GW-2, and at GW-4 are very similar, and therefore are not shown. Figure 3 shows that the query traffic is broken into two categories, ie. hash/urn based and non-hash/urn based queries. In non-hash/urn queries, the queries contain the simple ascii file name, while in hash based queries, each file is identified by a hash value as described in [3]. Overall among the outgoing traffic, query traffic dominates query hits traffic. This distribution suggests for passive Gnutella servents, the major saving from query caching will not be from reduced query hit messages, but from reduced query request messages themselves.

3.2.2 TTL Distributions

Figures 4 and 5 plots the TTL distributions of queries going through each of the eight servents as well as each of the three GWs. Figure 4 shows that the number of queries decreases experientially with increasing TTLs. This "exponential" behavior can be explained by the flooding scheme used by the Gnutella protocol: if all servents have the same number of neighbors, the number of servents a query will reach with decreasing TTLs grows exponentially. Conversely, if every servent initiates the same number of queries, the number of queries that reach a particular servent increase exponentially with decreasing TTLs.

Note that on average between 60% to 70% of all queries that reach each servent have $ttl = 1$ ($ttl = 0$ after decrementing), and these queries will not be forwarded further.

The queries that reach servents with $ttl > 0$ after decrementing will be forwarded to the GWs, and recorded by the tunneling probes. Our tunneling probes do not alter the TTL, and simply forward the queries to the destination servents outside. Figure 5 shows the TTL distribution of queries going out of the GWs, i.e., on hijacked outgoing connections. As expected, the percentage distribution is similar to the distribution of queries with $ttl > 0$ recorded at the individual servents, normalized after removing queries with $ttl = 0$.

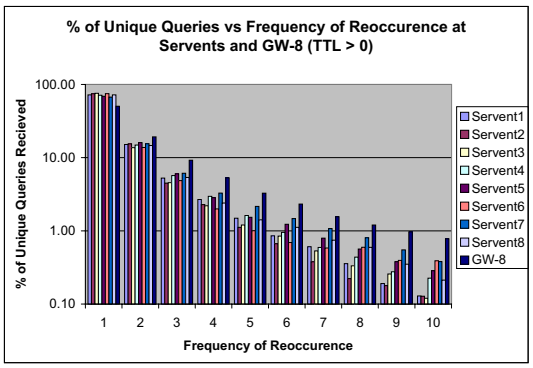

Figure 6. Query locality at individual servents going into the servents on their incoming and outgoing connections for queries with $ttl > 0$ after decrementing, excluding empty queries; and query locality at GW-8 for queries with $ttl > 0$ going into GW-8 on hijacked outgoing connections, excluding empty queries. Duplicates with the same `muid` are discounted.

Node	Weighted Frequency of Reoccurring
Servent 1 (unique (query str,ttl))	1.81
Servent 2 (unique (query str,ttl))	1.52
Servent 3 (unique (query str,ttl))	1.58
Servent 4 (unique (query str,ttl))	1.83
Servent 5 (unique (query str,ttl))	1.92
Servent 6 (unique (query str,ttl))	1.66
Servent 7 (unique (query str,ttl))	1.98
Servent 8 (unique (query str,ttl))	1.80
GW-2 (unique (query str, ttl, nbr))	1.67
GW-4 (unique (query str, ttl, nbr))	1.70
GW-8 (unique (query str, ttl, nbr))	1.79
GW-2 (unique (query str,ttl))	1.88
GW-4 (unique (query str,ttl))	2.33
GW-8 (unique (query str,ttl))	3.32

Table 1. Summary of locality in queries with $ttl > 0$ going into each servent on incoming and outgoing connections, and queries going into the three GWs on hijacked outgoing connections, both excluding empty queries. Duplicates with the same `muid` are discounted.

3.2.3 Locality at Each Servent

Before looking at the locality among the queries going through each GW, we first look at the locality among the queries going through each servent, and we separate the locality measurement for queries with $ttl = 0$ after decrementing, from those with $ttl > 0$ after decrementing, as the former category will not benefit from any form of caching, whether caching is performed at the servents or at the gateway. Figure 6 shows the percentage of unique queries as a function of the frequency they are witnessed by each of the eight servents, for $ttl > 0$ after decrementing. We see that the number of unique queries decreases close to exponentially as a function of their frequency of occurring, showing a popularity pattern similar to the Zipf-like distribution.

Table 1 lists the locality (i.e., weighted average frequency of reoccurring) of queries with $ttl > 0$ going into each servent and the three GWs. At each servent, the locality is measured by treating queries with unique query strings and TTLs as distinct queries.

3.2.4 Locality at the Gateway

Figure 6 also shows the percentage of unique queries as a function of the frequency that are witnessed by GW-8, the main router which hijacks all outgoing connections and thus witnesses all queries on those connections. We see that the percentage of high frequency queries is significantly higher than its counterparts at individual servents, suggesting that there is significant locality among the queries forwarded by the eight servents.

Note that if a servent has several outgoing connections to outside servents, it will forward a query through each of the connections. Furthermore, different servents can forward the same query (i.e., with identical `muid`) to outside because they share the same servent several hops back along the propagation tree of the query. Such repeated copies of the same query going through the gateway should not be counted as locality, since responses from one neighbor should not be used to serve a request to another neighbor. To avoid counting such repeated queries, our tunneling probe records the `muid` of each query it has seen, and discounts any queries whose `muid` has been encountered before,

Table 1 also lists the locality of queries with $ttl > 0$ going into each of the three GWs. The first set of locality numbers for GWs are counted by treating queries with unique query strings, TTLs, and neighbors as distinct queries. Thus, they do not contain locality among queries from different servents. When the servents do not share any neighbors outside the gateway, as is the case in our measurements, the locality at the GWs is similar to the locality at each of the servents. The second set of locality numbers for GWs are counted by treating queries with unique query strings and TTLs as distinct queries. Thus these numbers contain the locality among queries from different servents. Table 1 shows that the locality increases with the population of the servents. We conjecture the locality among the queries forwarded by servents within a large organization such as a university will be much higher.

4 Transparent Query Caching

The locality measurements in the previous section suggest that caching at the gateway of an organization can be far more effective than caching at individual servents behind the gateway because it can exploit the aggregated locality among all queries forwarded and initiated by all servents

inside the organization. In this section, we present a query caching scheme at the gateway of an organization. To simplify the discussion, we ignore caching HTTP downloads for now, and delay the discussion of integrating transparent query caching with transparent HTTP caching at the gateway till Section 5.

4.1 Overview

Our transparent caching scheme is similar to the locality measurement setup discussed in Section 3.1. The only difference is in the caching setup; the tunneling probe is replaced with a caching proxy that caches query responses (see Figure 1).

All the PING/PONG messages initiated or forwarded by the servents inside or outside going through the gateway will be forwarded by the proxy servent. The caching proxy will not change the TTL, and thus the reachability of PING/PONG messages remains the same as before. Similarly, HTTP data download messages will be tunneled through.

For query requests going out of the gateway, the caching proxy checks the local query hits cache. If it can serve query responses out of its cache, it will not forward the query request to outside the gateway. If it cannot serve query responses out of its cache, it will forward the query to the original destination of the query, without decrementing the TTL. Again, the reachability of this query stays the same as if the proxy does not exist.

4.2 The Caching Algorithm

As discussed in Section 3.2.4, the proxy servent attached to the gateway can potentially see repeated copies of the same query, i.e., with an identical `muid`. This complicates the notion of cache misses and cache hits in our transparent caching scheme. In the following, we discuss design options for the caching algorithm and present the details of the chosen algorithm.

4.2.1 Design Options

Each Gnutella query request tunneled by the caching proxy has a unique set of `muid`, query string, forwarder (inside the gateway), neighbor (outside the gateway), TTL, and minimum speed values. This is because repeated queries with the same `muid` have already been dropped at individual servents. In the following, we discuss design options in terms of the subsets of these parameters to be used for indexing query hits. Obviously, the smaller the subset, the more frequent the reuse of the cached query hits, i.e., the higher the cache hit ratio.

Out of the six parameters, query string and TTL are two obviously necessary caching parameters. Parameter `muid` should not be a parameter, because otherwise, queries with the same string and other parameters but different `muid` will never generate a hit. Thus the strictest caching algorithm is to index query hits by the tuple `(query string, forwarder, neighbor, ttl, minimum speed)`.

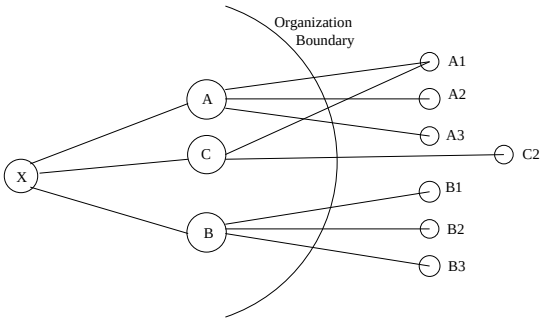

Figure 7. An example topology of servents inside the gateway.

Our caching algorithm further loosens up two parameters from the above tuple of five. First, it ignores the forwarder value, since query hits from a subtree (e.g., rooted at node $A1$ in Figure 7) due to a query sent by one forwarder (e.g., servent A) can be used to reply to a subsequent query with the same query string but from a different forwarder (e.g., servent C).

Second, it removes the minimum speed from the cache indexing tuple as follows. When a cache miss happens, the proxy rewrites the minimum speed to zero before forwarding it to outside the gateway. As a result, it collects query hits with all possible speeds. For subsequent queries that match all other parameters with the cached query hits, but specify a non-zero minimum speed requirement, the proxy can always extract a subset of the cached query hits that satisfy the minimum speed requirement without forwarding the query out of the gateway again. We call this scheme *one-time bandwidth adjustment*. Alternatively, the proxy can incrementally forward queries with lower and lower minimum speed requirement and thus could not be satisfied by previously cached query hits. In this case, the proxy replaces the cached query hits with newly received query hits every time it forwards a query with a lower minimum speed requirement. We call this scheme *incremental bandwidth adjustment*. Obviously, there is a tradeoff between the two schemes. The one-time bandwidth adjustment scheme pays a one-time cost to get all possible query hits, which may be higher than necessary, but can be used to reply all future recurrence of the same query. The *incremental bandwidth adjustment* scheme fetches query hits on demand, but it incurs overhead from repeated query hits when forwarding the same queries with lower and lower minimum speed requirements. We experimentally compare these two schemes in Section 4.3.

Case	Muid	Forwarder	Proxy Behavior
Case 1	equal	equal	will not happen
Case 2	equal	not equal	ignore and drop the query
Case 3	not equal	***	reply from the cache and drop the query

Table 2. Behavior of the caching algorithm for different cases.

A more aggressive caching algorithm further ignores the neighbor parameter and caches query hits indexed by `(query string, ttl)` only. This caching scheme can exploit the locality in the collective queries by different servents (see Table 1), since it can use query hits in its cache collected from replies from nodes in a subtree rooted at one neighbor outside the gateway (e.g., servent $A3$ in Figure 7) to reply to a query destined for a different subtree outside the gateway (e.g., rooted at servent $B3$). It remains unclear to what extent this scheme preserves the user searching experience as different subtrees may generate different sets of query hits for the same query with the same TTL. We are studying this caching scheme in our ongoing work.

4.2.2 The Chosen Algorithm

Our caching algorithm caches query hits according to the tuple of `(query string, neighbor, ttl)` values of the query. It uses two main data structures. First, any time the proxy tunnels a query to outside servents, it records the `muid`, the query string and the TTL information in a Cache Miss Table (CMT). When a query hit is received from outside, its *muid* is checked against CMT to find the corresponding query string and TTL, which is used to index into the cache table. The cache table (CT) is the data structure for storing cached query hits. For each CT entry CT(i), the algorithm adds a vector field to remember the `muid` and `forwarder` of up to 10 most recent queries for which query hits are replied using that cache entry.

Assume query hits for a particular query have been cached in the i_{th} entry of the cache, indexed by `(query string, neighbor, ttl)`. Next, when a query message MSG-1 results in a cache hit on this entry for the first time, the proxy replies from the cache and stores the `(muid, forwarder)` information of MSG in CT(i). Every time there is a cache hit with the i_{th} entry, the `(muid, forwarder)` information of the new message MSG-N is compared with those stored in CT(i) and the appropriate action according to Table 2 is taken.

- Case 1: The `(muid, forwarder)` of MSG-N match those of one of the previous messages replied by the proxy using the same cache entry. This will not happen as the forwarding servent should have dropped it.
- Case 2: The `muid` of MSG-N matches that of one of the previous messages replied by the proxy using the same cache entry, but their `forwarder` values differ. This only happens when a query is forwarded through different forwarders towards the same neighbor. In this case, the proxy must have already replied once to a query with the same `(muid, forwarder)` using cached query hits coming from the Gnutella subtree rooted at the cache entry's neighbor, and thus the proxy should simply drop MSG-N.
- Case 3: The `muid` of MSG-2 does not match with the `muid` of any of the previous queries replied by the proxy using cached query hits. In this case, the proxy is seeing this query message with this `muid` for the first time. Hence the proxy replies from the cache. The proxy also stores the `(muid, forwarder)` values of MSG-2 in the corresponding cache entry.

Optimization A simple optimization to our caching algorithm is to check CMT for the existence of an entry with the same `(muid, neighbor)` and drop the new query right away if such an entry is found. The reason is that the neighbor servent will drop the query anyway according to Gnutella protocol.

Caching delay A complication to our caching scheme rises when repeated queries (i.e., with the same cache index) are arriving at the proxy before all the query hits from outside the gateway for the first forwarded query have been received. We solve this problem as follows. Once there is a cache miss and the query is forwarded by the proxy, the proxy creates the corresponding entry, records the time, and declares it to be *unusable* until a certain period of time has elapsed, at which point, the cache entry is marked as *usable*. Our measurements at the caching proxy have shown that over 99% of the queries forwarded to outside the gateway receive all of their query hits within 15 seconds. Therefore we set the elapsed period to be 15 seconds. A hit on an unusable cache entry is treated the same as cache miss.

4.3 Caching Results

We ran two sets of experiments for an hour consecutively, one with one-time bandwidth adjustment and the other with incremental bandwidth adjustment. The caching results are shown in Table 3.

Table 3 shows the caching results for only non hash/urn based query traffic. We believe that the results will be similar when hash/urn based queries are also cached because the proposed caching algorithm will behave exactly the same way with hash/urn based queries as it does with non hash/urn based queries. One question about the caching results is whether to count the dropped queries due to optimization in Section 4.2.2 in calculating the hit ratios. Since

Cache version	One-time BW Adjustment	Incremental BW Adjustment
Time measured (EST)	3:30-4:30am, Jan 26, 2003 (after a 30-minute warmup)	2:00-3:00am, Jan 26, 2003 (after a 30-minute warmup)
number of non hash/urn queries (C1 + C2)	2386603	2150308
number of cache misses (C3)	1476160	1472218
number of cache misses generating nonempty query hits	10866	8545
number of cache misses due to speed	N/A	766
number of queries overwritten with 0 speed requirement	14669	N/A
number of cache hits, but unusable (Section 4.2.2 caching delay)	43003	45181
number of cache hits	903505	671441
number of cache hits with nonempty query hits	8939	11620
number of cache hits, but no reply (Table 2 Case 2)	2722	6702
number of queries dropped (C2) (Section 4.2.2 optimization)	6938	6649
query traffic from inside the gateway (B1) (KB)	197332.74	177651.66
query traffic tunneled from inside to outside the gateway (B2) (KB)	119259.12	118901.44
query hit traffic received from outside the gateway (B3) (KB)	36141.77	48383.82
query hit traffic sent by the proxy to inside the gateway (B4) (KB)	41263.61	51881.76
Measured cache hit ratios and byte hit ratios (with caching delay)		
query message hit ratio (C1+C2-C3)/(C1+C2)	38.15%	31.53%
query message byte hit ratio ((B1-B2)/B1)	39.56%	33.07%
query hit message byte hit ratio ((B4-B3)/B4)	12.41%	6.74%
Theoretical cache hit ratios (without caching delay)		
query locality (taking into account the neighbor and TTL)		
average frequency	1.664	1.510
query message hit ratio	(1.664 - 1)/1.664 = 39.90%	(1.510 - 1)/1.510=33.77%

Table 3. Caching results for the one-time bandwidth adjustment scheme and the incremental bandwidth adjustment scheme.

the dropped queries contribute to reducing the uplink traffic, we view them as cache hits.

The two versions of the caching algorithms result in 38.15% and 31.53% query message hit ratios (with dropped queries considered as hits), 39.56% and 33.07% byte hit ratios for query messages (i.e., reduction in uplink traffic), and 12.41% and 6.74% byte hit ratios for query hit messages (i.e., reduction in downlink traffic), respectively. Comparing the two caching algorithms, we see that the cache hit ratio in using the incremental bandwidth adjustment scheme is about 20% smaller than using the one-time bandwidth adjustment scheme. Since the number of cache misses due to a mismatch in speed requirement by using the incremental bandwidth adjustment scheme is rather insignificant (766 out of 1472218 misses), and the number of queries with non-zero speed requirements in the run using the one-time bandwidth adjustment scheme is also insignificant (14669 out of 1476160 misses), we attribute the difference in the hit ratios to the difference in locality in the two runs; the run using the one-time bandwidth adjustment scheme has higher locality than the run using the incremental bandwidth adjustment scheme. In other words, the two caching algorithms are likely to perform comparably given the same query traffic.

Table 3 also shows that there is only a small gap between the measured hit ratios which take into account the caching delay as explained in Section 4.2.2 and the theoretical hit ratios which are calculated based on the locality of queries assuming no caching delay. This suggests that repeated queries rarely happens right next to each other, and caching delay has little impact on the effectiveness of caching.

5 Integrating Query Caching with HTTP Caching

In this section, we present a scheme for integrating our query caching proxy with off-the-shelf HTTP caching proxies widely deployed in the Internet.

The fact that Gnutella servents often use the same TCP port for both protocol and data traffic poses challenges to transparently cache both types of traffic. On one hand, we cannot simply configure a regular web cache to listen on the default Gnutella port (i.e., 6346), as it will not understand Gnutella protocol messages. On the other hand, high performance off-the-shelf HTTP caching proxy are widely deployed, and thus it is desirable not to reimplement query caching and HTTP caching on top of one another.

Figure 8 illustrates our scheme for integrating transparent caching of Gnutella protocol and data traffic at the gateway of the organization. The router running WCCPv2 is configured to redirect TCP traffic destined to any specific port to the attached caching proxy machines. Two caching proxies are attached to the router: the first is our query caching proxy, listening on port 6346 only; the second is an off-the-shelf web caching proxy, listening on port 9346, assuming 9346 is an unused port. [2] The default configuration of the web caching proxy is changed slightly: on a

[2] We assume that additional web caching proxies listening on port 80 are attached to the router for caching normal web traffic.

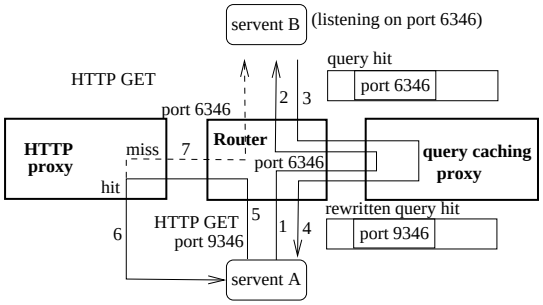

Figure 8. Integrating transparent query caching with HTTP data caching.

cache miss, instead of making a connection to port 80 of the origin server, it will make a connection to port 6346 of the origin server. Note that the two proxies can be running on the machine, as long as the router is properly configured. The major steps (labeled 1–6 in Figure 8) that Gnutella traffic are being redirected to the caching proxies and cached are as follows.

- The router redirects all protocol traffic going to port 6346 to the proxy cache (Step 1), and all HTTP traffic going to 9346 to the web caching proxy (Step 5). It will become clear below all Gnutella HTTP request from inside the gateway will not go to port 6346.
- The host running the proxy cache uses IP firewall to redirect protocol packets forwarded from the router to port 6346 of localhost.
- The proxy cache listening on port 6346 accepts these protocol messages, and processes them accordingly, as described in Section 4.2.2. Query misses will be forwarded to the destination servent, with destination port 6346 (Step 2). For each query hit coming back from a servent outside (Step 3), if the specified port is 6346, the query caching proxy will modify the port to 9346 before sending back to the servent inside the gateway (Step 4). Note if specified port is not 6346, it will not be rewritten.
- When the servent inside the organization downloads files via HTTP, it will specify port 9346 (Step 5), as dictated by the query hits it has received. This guarantees that all Gnutella HTTP requests going out of the gateway will use port 9346.
- The router redirects the HTTP request to port 9346 to the web caching proxy.
- The web caching proxy node uses IP firewall to redirect the HTTP packets to port 9346 of localhost.
- The web caching proxy accepts the HTTP requests, and proceeds as usual. On a hit, it replies to the servent out of its cache (Step 6). On a miss, it always makes a connection to the origin server on port 6346 (Step 7).

The origin server is guaranteed to be listening on port 6346. This is because only when the port specified in the query hit is 6346, the query caching proxy would rewrite it to 9346, as explained above. If specified port is not 6346, it would not have been overwritten, and the HTTP request would not have been redirected by the router. In other words, they would not be cached.

6 Related Work

Although caching peer-to-peer traffic is a relatively new topic, there have been many studies of the characteristics of Gnutella networks and traffic. Adar and Huberman studied the Gnutella traffic for a 24-hour period [4], and found that close to 70% of the users shared no files, and that 50% of all responses were returned by only 1% of the hosts. In [8], Saroiu et al. also observed that small percentage of the peers appeared to have "server-like" characteristics: they were well-connected in the overlay network and they served a significant number of files. In [7], Ripeanu et al. studied the topology of the Gnutella network over a period of several months, and reported two interesting findings: (1) the Gnutella network shares the benefits and drawbacks of a power-law structure, and (2) the Gnutella network topology does not match well with the underlying Internet topology leading to inefficient use of network bandwidth.

Several previous work studied query caching in Gnutella networks. Sripanidkulchai [10] observed that the popularity of query strings follows a Zipf-like distribution, and proposed and evaluated a simple query caching scheme by modifying a Gnutella servent. The caching scheme proposed was fairly simple; it caches query hits solely based on their query strings and ignores TTL values. In [6], Markatos studied one hour of Gnutella traffic traces collected at three servents located in Greece, Norway, and USA, respectively, and found that on average each query with $ttl > 1$ is witnessed by each servent between 2.57 to 2.98 times (disregarding the TTLs), suggesting significantly locality among queries forwarded by each servent. Markatos also proposed a query caching scheme by modifying servents that caches query hits according to the query string, the forwarder where the query comes from, and the TTL. In summary, all previous caching schemes for Gnutella networks focus on individual servents, and are implemented by modifying the servents and thus rely on wide adoption of modified servents to become effective. In contrast, our scheme views all the servents within an organization as a whole, exploits locality among the collective queries, requires no change to individual servents, and therefore can be easily deployed.

Several recent work studied other p2p traffic. Leibpwitz et al. [5] studied one month of FastTrack-based [11] p2p traffic at a major Israeli ISP and found that majority of the

p2p files are audio files and the majority of the traffic are due to video and application files. They also reported significant locality in the studied p2p data files. Saroiu et al. [9] studied the breakdowns of Internet traffic going through the gateway of a large organization into web, CDN, and p2p (Gnutella and KaZaa) traffic. They focused on HTTP traffic, for which caching techniques are well-known. In contrast, this paper focuses on the p2p protocol traffic, and proposes a transparent caching scheme for query traffic as well as a scheme for integrating transparent query caching with transparent HTTP caching.

7 Conclusions

We have studied the locality in the collective queries going through a gateway forwarded by servents behind that gateway, and found that there is a significant locality in the collective queries, and the locality increases with the population of those servents. To exploit the locality, we have proposed a scheme for transparently caching query hits at the gateway. Our scheme does not require any modifications to the individual servents, and can exploit the locality in the collective queries going through the gateway. We have implemented a conservative caching algorithm that preserves user's experience by distinguishing query hits from different Gnutella subtrees outside the gateway; queries will result in similar query hits with or without the transparent caching running. If servents inside the gateway do not share any neighbors outside the gateway, our conservative caching algorithm does not benefit from the locality in the collective queries, and it will generate similar cache hit ratios as caching at the individual servents (e.g., inside the gateway).

Measurements of our transparent caching proxy in an experimental testbed of 8 Gnutella servents in a LAN has shown a query cache hit ratio of up to 38%, an uplink query traffic reduction of up to 40%, and a downlink query hit traffic reduction of up to 12% at the gateway.

We are pursuing several directions in our ongoing work. First, we are studying the effectiveness and the accuracy of the more aggressive caching algorithm discussed in Section 4.2.1 which does not distinguish the subtrees outside the gateway from which query hits come from. Second, our tunneling probe and caching proxy only hijacks outgoing connections of servents inside the gateway. In principle, however, incoming connections can be hijacked by a proxy servent at the gateway in a similar fashion. We plan to study reverse caching of queries at the gateway. Third, to measure the effectiveness of our caching scheme in the real world which will a large number of active servents, we plan to deploy our tunneling probe and our caching proxy in the Purdue Research and Development Network (RDN). The RDN is a shadow network of the campus network developed to allow for researchers, faculty, and students to have access to Gigabit Network connectivity throughout the campus for research, testing, and development. It can be configured to take over adjustable amount of network traffic from the main campus network. Fourth, we also plan to study other p2p network traffic such as KaZaa, and extend the developed caching schemes for Gnutella to these other p2p networks and applications.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work was supported by Cisco Systems through the University Research Program.

References

[1] The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.

[2] The Gnutella 0.6 protocol draft, 2002. http://rfc-gnutella.sourceforge.net/.

[3] Hash/urn gnutella extensions (huge) v0.94, 2002. Gnutella Developer Forum.

[4] E. Adar and B. Huberman. Free riding on gnutella. *First Monday*, 5(10), 2000.

[5] N. Leibpwitz, A. B. an Roy Ben-Shaul, and A. Shavit. Are file swapping networks cacheable? characterizing p2p traffic. In *Proceedings of the 7th Intl. WWW Caching Workshop*, August 2002.

[6] E. P. Markatos. Tracing a large-scale peer to peer system: an hour in the life of gnutella. In *Proceedings of the 2nd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid 2002*, May 2002.

[7] M. Repeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[8] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, 2002.

[9] S. Saroiu, P. K. Gummadi, R. J. Dunn, S. D. Gribble, , and H. M. Levy. An analysis of internet content delivery systems. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[10] K. Sripanidkulchai. The popularity of gnutella queries and its implication on scaling, 2001.

[11] K. Truelove and A. Chasin. Morpheus out of the underworld, 2001.